



UNIVERSITÀ
DEGLI STUDI
FIRENZE

FLORE

Repository istituzionale dell'Università degli Studi di Firenze

Session and Union Types for Object Oriented Programming

Questa è la Versione finale referata (Post print/Accepted manuscript) della seguente pubblicazione:

Original Citation:

Session and Union Types for Object Oriented Programming / L. Bettini; S. Capecchi; M. Dezani-Ciancaglini; E. Giachino; B. Venneri.. - STAMPA. - (2008), pp. 659-680. [10.1007/978-3-540-68679-8_41]

Availability:

This version is available at: 2158/319871 since: 2017-05-26T11:19:46Z

Publisher:

Springer

Published version:

DOI: 10.1007/978-3-540-68679-8_41

Terms of use:

Open Access

La pubblicazione è resa disponibile sotto le norme e i termini della licenza di deposito, secondo quanto stabilito dalla Policy per l'accesso aperto dell'Università degli Studi di Firenze (<https://www.sba.unifi.it/upload/policy-oa-2016-1.pdf>)

Publisher copyright claim:

(Article begins on next page)

Abstract. In network applications it is crucial to have a mechanism to guarantee that communications evolve correctly according to the agreed protocol. Session types offer a method for abstracting and validating structured communication sequences (sessions). In this paper we propose union types for refining and enhancing the flexibility of session types in the context of communication centred and object oriented programming. We demonstrate our ideas through an example and a calculus formalising the main issues of the present approach. The type system guarantees that, in well-typed executable programs, after a session has started, values sent and received will be of the appropriate type, and no process can get stuck forever.

Keywords: Sessions, Object Oriented Programming, Session Types, Union Types

1 Introduction

Writing safe communication protocols has become a central issue in the theoretical practice of concurrent and distributed computing. The actual standards still leave to the programmer much of the responsibility in guaranteeing that the communications evolve as agreed by all the involved agents.

Session types [26, 27] offer a method for abstracting and validating structured communication sequences (*sessions*). This is achieved by giving types to communication channels, in terms of the types of values sent or received, e.g., the type $?int$ expresses that an integer will be received and then a boolean value will be sent. A session involves channels of dual session type, thus guaranteeing that, after a session has started, the values sent and received will be of the appropriate type. Since the typing of a session is a type, the conformance test of programs with respect to sessions becomes type checking.

The popularity of class-based *object oriented* languages justifies the interest in searching for class definitions which naturally include communication primitives.

* This work has been partially supported by MIUR project EOS DUE and by the Italian Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract 3-016004-IP-09).

Union types have been shown useful for enhancing the flexibility of subtyping in various settings [1, 21, 11, 10, 29]. For example a bank can answer yes or no depending on the balance between an account and an item price. If yes and not are the only possible answers, then the class of the object answering is the union of the two classes OK and NoMoney respectively, *i.e.* $OK \vee \text{NoMoney}$. With union types typing answer would require a superclass of both OK and NoMoney to be defined, and this superclass could include unwanted objects. With union types we can express communications between parties which manipulate heterogeneous objects by sending and receiving objects which belong to subclasses of one of the classes in the union. In this way the flexibility of object-oriented depth-subtyping is enhanced, strongly improving the expressiveness of choices based on the classes of sent and received objects.

The aim of the present paper is to discuss and formalise the use of union types in session-centred communications in a core object-oriented calculus. A preliminary version of the basic calculus, without union types, is defined in [18]. In the present paper the calculus of [18] is formally revised, so that typing and semantics are rather simple and simpler. Furthermore, the extension to union types, which is the main contribution of the present proposal, poses specific problems in formulating reduction and typing rules to ensure that communications are safe while flexible.

We first present an example which illustrates the main features of our approach, then we formalise these features through a featherweight representation. We call this language (Sessions Amalgamated with Methods plus union types) the language of the $\mathcal{F}\text{SAM}^\vee$ the formalising calculus.

SAM^\vee Overview. SAM^\vee , as the language of [18], is concerned with the amalgamation of the object oriented features with the session part, but it is agnostic w.r.t. to the other features of the language, such as whether the language is distributed or concurrent and the features for synchronisation.

In SAM^\vee , sessions and methods are “amalgamated”: invocation takes place on an object and the execution takes place immediately and concurrently with the receiving thread (indeed, SAM^\vee is multi-threaded and the communication is asynchronous). Thus, it keeps the method-like invocation mechanism while involving two threads, instead of session based communication mechanisms. The body is determined by the receiving object (avoiding in this way the usual branch/select primitive). A large number of communications interleaved with computation is possible. Methods are defined in a class, which can have also fields. We believe that the above model of session naturally reflects our intuition of services. Furthermore, it neatly encode “standard” methods.

A thread can make a session request through $e.s\{e'\}$, where e is an expression denoting an object, s is the name of a session defined in the object’s class; e' is evaluated to an object o , and the session body of s in o ’s class is executed concurrently with e' , introducing a new pair of fresh channels k and \tilde{k} (one for each communication).

The expressions $\text{send}(e)$ and $\text{rec}(x)$ send and receive objects on the active channel, respectively. The expression $\text{sendC}(e)\{C_1 \Rightarrow e_1 \parallel \dots \parallel C_n \Rightarrow e_n\}$ (where C means *Class*) evaluates e to an object and sends it on the active channel, and then continues with the choice C_i where C_i is the class that best fits the class of the object sent. The counterpart of sendC is the expression $\text{recC}(x)\{C_1 \Rightarrow e_1 \parallel \dots \parallel C_n \Rightarrow e_n\}$, where the choice is based on the class of the object received. The expression $\text{sendW}(e)\{C_1 \Rightarrow e_1 \parallel \dots \parallel C_n \Rightarrow e_n\}$ (where W means *While*) is similar to $\text{sendC}(e)\{C_1 \Rightarrow e_1 \parallel \dots \parallel C_n \Rightarrow e_n\}$, except that it is used for enclosed *cont*, which continues the execution at the nearest enclosing *send*. The counterpart of sendW is the expression $\text{recW}(x)\{C_1 \Rightarrow e_1 \parallel \dots \parallel C_n \Rightarrow e_n\}$ has the obvious meaning. Finally, the expression $\text{delegate}(s)$ delegates the current session to the object resulting from the evaluation of e . The session s of the session s in the class of that object is executed concurrently, using the same channel as the session. At the end, the final value of the body is passed to the current thread.

Related Papers. We describe $\mathcal{F}\text{SAM}^\vee$ following *Featherweight Java* [30], which has become a standard for class based object calculi.

Session Types have been first introduced to model communication protocols for π -calculus processes [26, 32, 27]. They have been made more expressive by equipping them with correspondence assertions [3], subtyping [24], bounded polymorphism [23] and safer by assuring deadlock-freedom [14]. More recently session types have been extended to multi-party communications [2, 9].

Session types have been developed also for CORBA [33], for functional languages [25, 34], for boxed ambients [22], for the W3C standard description language XML Schema Services CDL [8, 35, 31, 28], for operating systems [19], and for object oriented programming languages [17, 16, 13, 15, 18, 6]. In [6] generic types are added to a lambda calculus based on the approach of [18]; independently from the different type systems, $\mathcal{F}\text{SAM}^\vee$ improves the definition of the calculi of [18] and [6], both in syntax and in operational semantics.

Union types have been proved useful for functional languages [1, 11], for object oriented languages [29], for languages manipulating semi-structured data [2] and for the π -calculus [10]. We will tell more on the relations between the present work and [29] at the end of Subsection 6.1.

There are many concurrent object-oriented languages and calculi in the literature. In this topic we refer to the related work section of [20].

Paper Structure. In Section 2 we describe SAM^\vee in terms of an example. In Section 3 we proceed by formalising the calculus $\mathcal{F}\text{SAM}^\vee$, its typing and semantics. Section 4 discusses some future work directions.

2 An Example

In this section, we describe SAM^\vee through an example, which expresses a type of laboration pattern, *c.f.* [35, 7, 8], and which refines the example of [18]. The

¹ k and \tilde{k} play for channels a role similar to that of *this* for objects.

```

4      MakeAnOffer  $\Rightarrow ? \{ \text{Money} \Rightarrow \alpha, \text{NoDeal} \Rightarrow \varepsilon \}$ 
5
6      sessiontype ExaminePrice_ST = ?Money.!Ok\NoDeal\MakeAnOffer
7
8      sessiontype Sell_ST =
9          ?Item.!Money. $\mu\alpha.?$ { OK  $\Rightarrow$  ?AccntNr.?Money.!{ OK $\Rightarrow$ !Date, NoMoney  $\Rightarrow$ 
10              NoDeal  $\Rightarrow \varepsilon$ ,
11              MakeAnOffer  $\Rightarrow$  ! { Money  $\Rightarrow \alpha$ , NoDeal  $\Rightarrow \varepsilon$  }
12
13      sessiontype CalDelDate_ST = ?Item.!Date
14
15      sessiontype CalNewPrice_ST = ?Money.!Money\NoDeal
16
17      sessiontype CreditCheck_ST = ?AccntNr.?Money

```

Fig. 1. Session types for the buyer-seller example

protocol contains essential features which demonstrate the expressiveness of SAM^\vee . The buyer negotiates a price from a seller, and if and when reached agreement, he sends his bank account number so that it gets verified he has enough money. If he has enough money, he receives the delivery date, otherwise the deal falls through. The seller *delegates* to a bank the part of the session that concerns money in the account. Such delegation has traditionally been expressed through nested order sessions; instead, SAM^\vee can delegate the current session through a session type in [18,6]. The negotiation allows several rounds: the buyer may either accept the offer or break the negotiation, or require a better deal by sending different kinds of objects. In the latter case, the seller might respond by sending a better price, or might refuse the negotiation sending a negative answer. Thus, branch selection in control structures is based on the dynamic class of an object sent.

The session types Shopping_ST and Sell_ST (see Figure 1) describe the communication pattern between the Buyer and the Seller.

The session type Shopping_ST describes the above protocol from the point of view of the buyer. The part !Item.?Money indicates sending an Item followed by a Money. The recursive type $\mu\alpha. \{ \text{OK} \Rightarrow \dots, \text{NoDeal} \Rightarrow \dots, \text{MakeAnOffer} \Rightarrow \dots \}$ describes the negotiation part, whereby an object is sent, and then, depending on whether the actual object sent belongs to class OK, NoDeal, or MakeAnOffer, either the second or third branch is taken. In the first branch, the account number and the delivery date are sent; then, either OK followed by a Date, or a NoMoney is received. In the third branch, a further object is received, and if that object is a Money, then the negotiation continues on the basis of it, whereas if it is a NoDeal, the negotiation ends.

Note that in both Shopping_ST and Sell_ST the recursion variable is nested under multiple choices, so that this behaviour could not have been expressed using simple expressions as in [15]. The use of recursive types has also other advantages, such as allowing iterative expressions with multiple exit points and multiple recursive calls.

```

4   String\Date Shopping_ST shopping
5   { Item prodId := ....;
6     seller.sell{
7       send(prodID);
8       Money price := rec;
9       examinePrice{send(price);
10          OK\NoDeal\MakeAnOffer resp := rec};
11      sendW(resp){
12          OK ⇒ { send(accnt); send(price);
13                recC(x) { OK ⇒ Date delivDate:=rec; []
14                  NoMoney ⇒ new String("no money") []
15                  NoDeal ⇒ new String("refusing proposed price") []
16                  MakeAnOffer ⇒ {
17                      recC(x) { Money ⇒ examinePrice{send(x);
18                          resp := rec};
19                          cont; []
20                          NoDeal ⇒ new String("offer re
21                      }
22                  } //end of session call sell
23              } //end of session shopping
24
25  Object ExaminePrice_ST examinePrice
26      { Money price := rec
27        ... //code for decision
28        send(resp)
29      } // end of session examinePrice
30  }

```

Fig. 2. The class Buyer

Figure 2 we show the implementation of the class Buyer. It has the fields a seller, which will contain the account number and the seller used to buy

The class Buyer supports two sessions called shopping and examinePr sion shopping has session type Shopping_ST and return type String\Date union type String\Date describes the possible results of the negotiation: success the session ends returning the date of the delivery of the item; in case it returns a string describing the reason. In the body of this session the desired is determined and stored in prodId (line 5). Then, a session request is made seller to run session sell (line 6). Thus, the seller will run the body of parallel with the remaining part of the session body of shopping, and a connection be created between the two threads. On this connection, the buyer will send (line 7), receive a Money and store it in price (line 8). Based on its value he will calculate his response calling session examinePrice which returns the resp (lines 9 and 10). Let us notice that resp's type is the union of all the answers' type. On line 11, the buyer enters a loop with sendW, where he sends and branches according to its class. If resp is OK, indicating acceptance of

```

4      String\Item Sell_ST sell
5      { Item prodID := rec;
6        Money price=...;
7        send(price);
8        recW( x ){
9          OK ⇒ { sendC( bank•check{ } )
10                {OK ⇒ calDelDate{send(prodID);
11                      Date date := rec};
12                      send(date); prodID []
13                NoMoney ⇒ new String("failed bank transac
14                      } } []
15          NoDeal ⇒ new String("proposal refused by buyer") []
16          MakeAnOffer ⇒ { calNewPrice{send(price);
17                          Money\NoDeal resp := rec} ;
18                          sendC(resp){ Money ⇒ cont []
19                                NoDeal ⇒ new String("re
20                                proposed price"); }
21                          }
22          } //end of session sell
23
24      Object CalDelDate_ST calDelDate
25      { Item item := rec
26        ... //code for calculate date
27        send(date)
28      } // end of session calDelDate
29
30      Object CalNewPrice_ST calNewPrice
31      { Money price := rec
32        ... //code for response (if the answer is positive then the
33          //price is updated with the new price)
34        sendC(resp)
35      } // end of session calNewPrice
36    }

```

Fig. 3. The class Seller

then the buyer will send his account number, and price (line 12); and will receive a response object which may be OK, in which case he will receive a Date and store it in date (line 13), or will receive a NoMoney (line 14). In this case the reason of failure is stored in the string failure. If resp is NoDeal, indicating that the price is unacceptable, then the session terminates. If the response is MakeAnOffer, inviting the seller to make a better offer, then the rest depends on the other party's response, indeed the session contains recC indicating that a value will be received, and the remaining steps are determined by its class. If the value received is a Money then session examination is called, which returns the buyer's reaction in resp, and the recursion will continue (line 19). If the value received is NoDeal, then the loop will be abandoned.

```

4      Money amt := rec;
5      .... // code for check
6      If(response) then new Ok else new NoMoney;
7      } // end of session check
8  }

```

Fig. 4. The class Bank

Notice that in order to get an arbitrary number of repetitions, it is crucial to have objects of different classes to be sent in the different iterations of while loops.

The session type `Sell_ST` describes the protocol from the point of view of the seller, and is “dual” to `Shopping_ST`. We now consider the class `Seller` in Figure 3. The session body for `sell` starts by receiving the description of the item, calculating and sending its price. Then, in line 8, it enters a `recW` loop, which is the counterpart to the `sendW` loop from `shopping` and performs all the seller’s actions. The interesting feature shown here is *delegation*, on line 9, whereby, the seller is requested to continue the session, using the current connection, and by applying the session body check. At the end of the execution of `check` the session will continue according to the bank answer (OK or NoMoney).

The session type for `check` from class `Bank` in Figure 4 is the receipt of a `Money` and a `Money` followed by sending either OK, or a `NoMoney` object. Note that the session body for `check` is *not aware* whether it will be called through a session request or through delegation. The return type of `check` is the union of the types of the possible answers, *i.e.* $OK \vee \text{NoMoney}$.

Notice that the sessions `examinePrice`, `calNewDate` and `calNewPrice` are examples of the implementation in SAM^\vee of methods, since they start by receiving a request and after elaborating them send a result.

3 Syntax

This section presents the syntax of $\mathcal{F}\text{SAM}^\vee$ (Figure 5), a minimal concurrent imperative core calculus, based on *Featherweight Java* [30] (abbreviated FJ). $\mathcal{F}\text{SAM}^\vee$ supports the basic object-oriented features and session request, session delegation, branching sending/receiving and loops. In details, $\mathcal{F}\text{SAM}^\vee$ encompasses the following linguistic features: basic object oriented expressions, session bodies, session communication constructs that combine send/receive with branching and loops.

We use grey to indicate expressions that are produced during the reduction process but do not occur in the source code of a program. We also use the standard notation of denoting with $\bar{\xi}$ a sequence of elements ξ_1, \dots, ξ_n .

Union types are defined as in [29]: they are built out of class names by the union operator (denoted by \vee).

Programs are defined from a collection of classes. The metavariables C and c range freely with subscripts, range over class names. Each class has a name, a list of fields

(expression) $e ::= x \mid \text{this} \mid \text{cont} \mid o \mid e; e \mid e.f := e \mid e.f \mid \text{new}$
 $\mid e.s\{e\} \mid e \bullet s\{k\}$
 $\mid k. \text{sendC}(e)\{C \Rightarrow e \parallel C \Rightarrow e\}$
 $\mid k. \text{recC}(x)\{C \Rightarrow e \parallel C \Rightarrow e\}$
 $\mid k. \text{sendW}(e)\{C \Rightarrow e \parallel C \Rightarrow e\}$
 $\mid k. \text{recW}(x)\{C \Rightarrow e \parallel C \Rightarrow e\}$
(parallel threads) $P ::= e \mid P \parallel P$

Fig. 5. Syntax, where syntax occurring only at runtime appears shaded. Syntax for τ is in Figure 9.

form \overline{Tf} , where f represents the field name and T its type, and a list of *session* forms $T\tau s\{e\}$, where T is the return type, τ the session type, s the session name, and e the session body. For the sake of conciseness the symbol \triangleleft represents class inheritance as in [30]. All classes are defined as extensions of the topmost class `Object`.

Expressions include variables, that are both standard term variables x and term variables `this` and `cont`. The variable `this` is considered implicitly bound in a session declaration. Instead, `sendW` and `recW` are the only binders for `cont`, that bind the continuation by recursive computation. Let us notice that free occurrences of e are not bound in the expression `sendW(e){...}`: actually no occurrence of e appear in e if this expression is typable (see rule `SENDW-T` in Figure 11).

In a *session request* $e.s\{e'\}$ we call the expression e' the *co-body* of the request (since it will be evaluated concurrently with the body of requested session).

In the *session delegation* expression, $e \bullet s\{k\}$, the channel k is added by the operational semantics in order to keep track of the channel to pass to the delegated session.

Channels are implicit in the source language syntax. At runtime, communication channels k are introduced at each new session request. We denote the dual of a channel k where \tilde{k} is again a runtime channel, and where \sim is an involution: $\tilde{\tilde{k}} = k$. When a thread uses a channel k , the other participant in the communication uses its dual \tilde{k} . In the operational semantics associates to k and \tilde{k} two different queues of messages. When a thread, which uses the channel k , wants to receive a message it will inspect the queue associated to k , while, when it sends a message it will add it to the queue associated to \tilde{k} (see Section 5).

The body of a *communication expression* is a pair of alternatives $\{C_1 \Rightarrow e_1 \mid C_2 \Rightarrow e_2\}$, whose choice depends on the class of the object that is sent or received. In particular, in case of `sendW` and `recW` the expressions e_i can contain `cont`, representing recursive computations.

With respect to SAM^\vee we only have binary choices as bodies of communication expressions in \mathcal{FSAM}^\vee , since the other forms can be encoded with them. First, the unary choice $\{C \Rightarrow e\}$ can be simply encoded as $\{C \Rightarrow e \parallel C \Rightarrow \text{new Object}\}$, since sending or receiving an object of class C will always choose the first alternative.

$\text{recC}(x)\{\text{Object} \Rightarrow x\}$ encodes rec .

With binary choices we can also encode n -ary choices for $n > 2$, getting in the constructs used in the example of Section 2. The informal idea of such given the set of classes to be used in the choices, is to take the first (in the le order) relative minimum from this set and use it to define the first branch; the first branch will use Object and we iterate this procedure to write the expression for the second branch, until we remain with only two choices. Thus, for instance, consider the choice $\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2 \parallel C_3 \Rightarrow e_3\}$, where C_1 is not related to the other classes and C_2 is a subclass of C_3 ; we can encode this choice with the (nested) binary choice: $\{\text{Object} \Rightarrow \{C_2 \Rightarrow e_2 \parallel C_3 \Rightarrow e_3\}\}$. Notice that this encoding is correct also if C_1 is a subclass of C_3 .

The types used for selecting branches in a choice are class names. This is the formal treatment and the proofs, but, again, we can encode choices with union types by n -ary choices in a straightforward way; e.g., $\{C_1 \vee C_2 \Rightarrow e \parallel C_3 \Rightarrow e'\}$ can be encoded as $\{C_1 \Rightarrow e \parallel C_2 \Rightarrow e \parallel C_3 \Rightarrow e'\}$.

A *runtime expression* is either a user expression (i.e. an expression in Figure 2, without shaded syntax) or an expression containing channels and/or object identifiers. Furthermore, threads of runtime expressions can occur at runtime (see the operational semantics). Parallel threads are ranged over by P . Fully evaluated objects will be denoted by object identifiers denoted by o .

The main novelty of \mathcal{FSAM}^V w.r.t. FJ is that session invocation can involve the creation of concurrent and communicating threads. Other minor differences are that we do not have cast and overriding, which are orthogonal to our approach; we do not have explicit constructors, then in the object instantiation expression $\text{new } C(\bar{e})$, the \bar{e} to which \bar{e} reduce are the initial values of the fields.

Notice that standard methods can be seen as special cases of sessions. A method declaration can be (informally) encoded as a session with nested requests (one for each parameter) and with one sendC returning the method body. Similarly, method calls are special cases of session requests: the passing of arguments is encoded by sendCs (one for each argument) and the object returned by the method body is received via one recC . This encoding will use unary choices that can be rendered with binary choices as explained above.

4 Auxiliary Functions

As in FJ, a class table CT is a mapping from class names to class declarations. The domain $\mathcal{D}(CT)$. Then a program is a pair (CT, e) of a class table (containing the class definitions of the program) and an expression e (an expression belonging to the source language representing the program's main entry point). The class Object has no fields and its declaration does not appear in CT . As in FJ, from any C we can read off the subtype relation between classes, as the transitive closure of \leq . Moreover this relation is extended in order to relate types built out of union types. As usual considering union types modulo the equivalence relation induced by \leq we get the commutativity and associativity of \vee . Therefore each union type can

$$\begin{array}{c}
\frac{}{T <: T \vee T'} \quad \frac{}{T' <: T \vee T'} \quad \frac{}{T' \vee T'' <: T} \\
\\
\frac{\text{fields}(\text{Object}) = \bullet \quad \frac{\text{fields}(D) = \overline{T'} \overline{F'} \quad \text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \}}{\text{fields}(C) = \overline{Tf}, \overline{T'} \overline{F'}}}{\text{fields}(C) = \overline{Tf}} \\
\\
\frac{}{\text{ftype}_w(\mathbf{f}_i, C) = \text{ftype}_r(\mathbf{f}_i, C) = T_i} \\
\\
\text{ftype}_w(\mathbf{f}, T_1 \vee T_2) = \begin{cases} \text{ftype}_w(\mathbf{f}, T_1) & \text{if } \text{ftype}_w(\mathbf{f}, T_1) <: \text{ftype}_w(\mathbf{f}, T_2), \\ \text{ftype}_w(\mathbf{f}, T_2) & \text{if } \text{ftype}_w(\mathbf{f}, T_2) <: \text{ftype}_w(\mathbf{f}, T_1), \\ \perp & \text{otherwise.} \end{cases} \\
\\
\text{ftype}_r(\mathbf{f}, T_1 \vee T_2) = \text{ftype}_r(\mathbf{f}, T_1) \vee \text{ftype}_r(\mathbf{f}, T_2) \\
\\
\frac{\text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \} \quad T \text{ ts } \{ e \} \in \overline{S}}{\text{stype}(s, C) = \{ \tau \}} \quad \frac{\text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \} \quad s}{\text{stype}(s, C) = \text{stype}(s, D)} \\
\\
\text{stype}(s, T_1 \vee T_2) = \text{stype}(s, T_1) \cup \text{stype}(s, T_2) \\
\\
\frac{\text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \} \quad T \text{ ts } \{ e \} \in \overline{S}}{\text{rtype}(s, C) = T} \quad \frac{\text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \} \quad s}{\text{rtype}(s, C) = \text{rtype}(s, D)} \\
\\
\text{rtype}(s, T_1 \vee T_2) = \text{rtype}(s, T_1) \vee \text{rtype}(s, T_2) \\
\\
\frac{\text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \} \quad T \text{ ts } \{ e \} \in \overline{S}}{\text{sbody}(s, C) = e} \quad \frac{\text{class } C \triangleleft D \{ \overline{Tf}; \overline{S} \} \quad s}{\text{sbody}(s, C) = \text{sbody}(s, D)}
\end{array}$$

Fig. 6. Subtyping and Lookup Functions

as $C_1 \vee \dots \vee C_n$ for $n \geq 1$: we say that the classes C_1, \dots, C_n *build* the union $\dots \vee C_n$. A union type $C_1 \vee \dots \vee C_n$ is *proper* if $n > 1$.

We assume a fixed *CT* that satisfies some usual sanity conditions as in FJ. Thus, in the following, instead of writing $CT(C) = \text{class } \dots$ we will simply write $\text{class } C \dots$

We define auxiliary functions (see Figure 6) to lookup fields and sessions: these functions are used in the typing rules and in the operational semantics. The lookup function is as in FJ. As for field type lookup we distinguish between those where the field is used for reading (ftype_r) from those where it is for writing (ftype_w). stype and rtype return a set of session types and the return type of a session, respectively, while sbody returns the body of a session. As in FJ these functions may have to traverse the class hierarchy in case the required element is not present in the current class.

Notice that the type lookup functions take a type as argument (not simply an expression) because the receiver expression of a field/session access may be of a union type. As for field type lookup, when the field is used in read mode, and the type is a proper union type, we simply return the union type of the result of ftype_r on the argument types (if both retrievals succeed). On the contrary, when the field is used in write mode, and the type is a proper union type, we return the union type updated, due to the contravariance relation, in case of a proper union type.

$$\begin{aligned}
e[k] &= \begin{cases} e_1[k].l : e_2[k] & \text{if } e = e_1.l : e_2, \\ e_1[k].s \{e_2\} & \text{if } e = e_1.s \{e_2\}, \\ e_1[k] \bullet s \{k\} & \text{if } e = e_1 \bullet s \{k\}, \\ k.\text{sendC}(e_0[k]) \{\overline{C} \Rightarrow e[k]\} & \text{if } e = \text{sendC}(e_0) \{\overline{C} \Rightarrow e\}, \\ k.\text{recC}(x) \{\overline{C} \Rightarrow e[k]\} & \text{if } e = \text{recC}(x) \{\overline{C} \Rightarrow e\}, \\ k.\text{sendW}(e) \{\overline{C} \Rightarrow e[k]\} & \text{if } e = \text{sendW}(e) \{\overline{C} \Rightarrow e\}, \\ k.\text{recW}(x) \{\overline{C} \Rightarrow e[k]\} & \text{if } e = \text{recW}(x) \{\overline{C} \Rightarrow e\}, \\ e & \text{otherwise.} \end{cases} \\
e[e'/\text{cont}] &= \begin{cases} e_1[e'/\text{cont}] ; e_2[e'/\text{cont}] & \text{if } e = e_1 ; e_2, \\ e_1[e'/\text{cont}].f & \text{if } e = e_1.f, \\ e_1[e'/\text{cont}].f := e_2[e'/\text{cont}] & \text{if } e = e_1.f := e_2, \\ e_1[e'/\text{cont}].s \{e_2\} & \text{if } e = e_1.s \{e_2\}, \\ e_1[e'/\text{cont}] \bullet s \{k\} & \text{if } e = e_1 \bullet s \{k\}, \\ k.\text{sendC}(e_0) \{\overline{C} \Rightarrow e[e'/\text{cont}]\} & \text{if } e = k.\text{sendC}(e_0) \{\overline{C} \Rightarrow e\}, \\ k.\text{recC}(x) \{\overline{C} \Rightarrow e[e'/\text{cont}]\} & \text{if } e = k.\text{recC}(x) \{\overline{C} \Rightarrow e\}, \\ e' & \text{if } e = \text{cont}, \\ e & \text{otherwise.} \end{cases}
\end{aligned}$$

Fig. 7. Channel Addition and Continuation Replacement

return the intersection of the result of $f\text{type}_w$ on the arguments; however, in the case of multiple inheritance, the only possible cases are those listed in Figure 6, thus avoiding introducing intersection types.

As for the $s\text{type}$ lookup function, it returns a set of session types; in case it is invoked with a class name as argument, it will return a singleton. The interesting case is when it is invoked with a proper union type: it will return the union of the sets corresponding to the argument types, so that we have all the session types of the classes in the union type (see how it is used in the typing system, Figures 11 and 13). The $s\text{type}$ lookup function behaves in a covariant way since the resulting object cannot be used in writing mode. We notice that $s\text{body}$ is only invoked with a class name as type argument since we invoke sessions on objects only, and all objects have a class name as type.

It is easy to verify that all lookup functions applied to equivalent union types return either equivalent union types or the same sets of session types, whenever they are

5 Operational Semantics

Objects passed in asynchronous communications are stored in a *heap*. A heap h is a mapping with domain consisting of objects and channel names. Its syntax is

$$h ::= [] \mid o \mapsto (C, \overline{f} : o) \mid k \mapsto \overline{o} \mid h :: h$$

where $::$ denotes heap concatenation.

by $o \mapsto (C, f : o)$.

The form $h[o \mapsto h(o)[f \mapsto o']]$ denotes the update of the field f of the object $h(o)$ to the object o' .

Channel names are mapped to queues of objects: $k \mapsto \bar{o}$. The heap projection $h[k \mapsto \bar{o}]$ maps the channel k to the queue \bar{o} . With some abuse of notation we write $o :: \bar{o}$ and $\bar{o} :: o$ to denote the queue whose first and last element is o , respectively.

Heap membership for object identifiers and channels is checked using standard notation, by identifying h with its domain, we can also write $o \in h$, and $k \in h$.

The values that can result from normal termination are parallel threads of terminated objects.

In the reduction rules we make use of the special *channel addition* operation $\bar{\cdot}$ and of the *continuation replacement* operation $[\dots/\text{cont}]$ (their formal definition is in Figure 7, where $\{\bar{C} \Rightarrow e\}$ is short for $\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}$). We denote by e a source expression in which all occurrences of receive, send, and delegation expressions which *are not* within the co-body of a session request are extended, so that they explicitly mention the channel k they will use (remember that channel names are chosen by the programmer). Also, we denote by $e[e'/\text{cont}]$ the expression e in which all occurrences of cont , that *are not* within the co-body of a session request or the co-body of a send/receive loop, are replaced by e' , thus preserving the correct nesting of while expressions. For example $\text{recC}(x)\{C_1 \Rightarrow x \parallel C_2 \Rightarrow \text{cont}\}\{k\}[e]$ becomes $k.\text{recC}(x)\{C_1 \Rightarrow x \parallel C_2 \Rightarrow e'\}$.

The reduction is a relation between pairs of threads and heaps:

$$P, h \longrightarrow P', h'$$

Reduction rules use evaluation contexts (based on runtime syntax) that capture the “next subexpression to be reduced”:

$$\begin{aligned} \mathcal{E} ::= & [-] \mid \mathcal{E}; e \mid \mathcal{E}.f \mid \mathcal{E}.f := e \mid o.f := \mathcal{E} \mid \mathcal{E}.s\{e\} \mid \\ & \mathcal{E} \bullet s\{k\} \mid k.\text{sendC}(\mathcal{E})\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} \end{aligned}$$

The explicit mention of the evaluation context is needed in rule SESSREQ-R (Figure 8) in which a new thread is generated in parallel with the evaluation context.

Reduction rules are in Figure 8. Rule PAR-R models the execution of parallel composition. In this rule parallel composition is considered modulo structural equivalence. To formalize this, we define structural equivalence rules asserting that parallel composition is associative and commutative:

$$P \parallel P_1 \equiv P_1 \parallel P \quad P \parallel (P_1 \parallel P_2) \equiv (P \parallel P_1) \parallel P_2 \quad P \equiv P' \Rightarrow P \parallel P_1 \equiv P' \parallel P_1$$

The successive four rules define the execution of standard object-oriented constructs.

Rule SESSREQ-R models the connection between the co-body e of a session request $o.s\{e\}$ and the body e' of the session s , in the class of the object o . This connection

$$\begin{array}{c}
\text{NEWC-R} \\
\frac{fields(C) = \overline{Tf} \quad o \notin h}{\mathcal{E}[\text{new C}(\overline{o})], h \longrightarrow \mathcal{E}[o], h :: [o \mapsto (C, \overline{f : o})]}
\end{array}
\quad
\begin{array}{c}
\text{FLDASS-R} \\
\mathcal{E}[o.f := o'], h \longrightarrow \mathcal{E}[o'], h[o \mapsto h(o)]
\end{array}$$

$$\begin{array}{c}
\text{SESSREQ-R} \\
\frac{h(o) = (C, _) \quad sbdy(s, C) = e' \quad k, \tilde{k} \notin h}{\mathcal{E}[o.s\{e\}], h \longrightarrow \mathcal{E}[e\{k\}] \parallel [o/\text{this}]e'\{\tilde{k}\}, h[k, \tilde{k} \mapsto ()]}
\end{array}
\quad
\begin{array}{c}
\text{SESSDEL-R} \\
\frac{h(o) = (C, _) \quad sbdy(o \bullet s\{k\}) = e'}{\mathcal{E}[o \bullet s\{k\}], h \longrightarrow \mathcal{E}[[o/\text{this}]e']}
\end{array}$$

$$\begin{array}{c}
\text{SENDCASE-R} \\
\frac{h(\tilde{k}) = \overline{o} \quad h(o) = (C, _) \quad C \Downarrow \{C_1, C_2\} = C_i}{\mathcal{E}[k.sendC(o)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}], h \longrightarrow \mathcal{E}[e_i], h[\tilde{k} \mapsto \overline{o} :: o]}
\end{array}$$

$$\begin{array}{c}
\text{RECEIVECASE-R} \\
\frac{h(k) = o :: \overline{o} \quad h(o) = (C, _) \quad C \Downarrow \{C_1, C_2\} = C_i}{\mathcal{E}[k.recC(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}], h \longrightarrow \mathcal{E}[[o/x]e_i], h[k \mapsto \overline{o}]}
\end{array}$$

$$\begin{array}{c}
\text{SENDWHILE-R} \\
\mathcal{E}[k.sendW(e)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}], h \longrightarrow \mathcal{E}[k.sendC(e)\{C_1 \Rightarrow e'_1 \parallel C_2 \Rightarrow e'_2\}], h
\end{array}$$

where $e'_i = \lfloor e_i / \text{cont} \rfloor k.sendW(e)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} / \text{cont}$

$$\begin{array}{c}
\text{RECEIVEWHILE-R} \\
\mathcal{E}[k.recW(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}], h \longrightarrow \mathcal{E}[k.recC(x)\{C_1 \Rightarrow e'_1 \parallel C_2 \Rightarrow e'_2\}], h
\end{array}$$

where $e'_i = \lfloor e_i / \text{cont} \rfloor k.recW(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} / \text{cont}$

Fig. 8. Reduction Rules

established through a pair of fresh channels k, \tilde{k} . For this purpose the expression $e\{k\}$ reduces, in the same context, to its own co-body $e'\{\tilde{k}\}$ and in parallel, outside the context, it spawns the body $[o/\text{this}]e'\{\tilde{k}\}$ of the called session. The explicit substitution of e and of \tilde{k} in e' ensures that the communication is on the fresh dual channel k, \tilde{k} . Thus, an object can serve *any number* of session requests. For example,

$$\begin{aligned}
& o.s\{\text{sendC}(5)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}\}; \text{new C}(_) \longrightarrow \\
& k.sendC(5)\{C_1 \Rightarrow e_1\{k\} \parallel C_2 \Rightarrow e_2\{k\}\}; \text{new C}(_) \parallel \\
& \tilde{k}.recC(x)\{C'_1 \Rightarrow [o/\text{this}]e'_1\{\tilde{k}\} \parallel C'_2 \Rightarrow [o/\text{this}]e'_2\{\tilde{k}\}\}
\end{aligned}$$

if $\text{recC}(x)\{C'_1 \Rightarrow e'_1 \parallel C'_2 \Rightarrow e'_2\}$ is the body of session s in the class of the object o . Notice that there is no ambiguity in this rule, since

$$\begin{aligned}
& (k.sendC(5)\{C_1 \Rightarrow e_1\{k\} \parallel C_2 \Rightarrow e_2\{k\}\} \parallel \\
& \tilde{k}.recC(x)\{C'_1 \Rightarrow [o/\text{this}]e'_1\{\tilde{k}\} \parallel C'_2 \Rightarrow [o/\text{this}]e'_2\{\tilde{k}\}\}); \text{new C}(_)
\end{aligned}$$

is not a thread according to the syntax of $\mathcal{F}\text{SAM}^\vee$.

Rule **SESSDEL-R** replaces the session delegation $o \bullet s\{k\}$ by $[o/\text{this}]e'$ where e is the body of the session s , in the class of the object o . This allows a parallel communication to be delegated via the channel k to the object o : this delegation

if $\text{recC}(\mathbf{x})\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\}$ is the body of session s in the class of the object o , *i.e.*, of evaluating the expression e , in the queue associated to the dual channel communication channel k . The computation then proceeds with the expression $C_1 \neq C_2$ and C_i is the smallest class in $\{C_1, C_2\}$ to which the object o belongs. If $C_1 = C_2$ and o belongs both to C_1 and to C_2 , then the computation proceeds with e_1 ². This is given by the condition $h(o) = (C, \perp)$ and by the following definition: $C \Downarrow \{C_1, C_2\} = C_i$, using the subtyping relation (Figure 6):

$$C \Downarrow \{C_1, C_2\} = \begin{cases} C_i & \text{if } C <: C_i \text{ and } C <: C_j \text{ with } i \neq j \text{ imply } C_j \not<: C_i, \\ C_1 & \text{if } C <: C_1 = C_2, \\ \perp & \text{otherwise.} \end{cases}$$

Dually the receive communication rule takes an object o from the queue associated to channel k and returns the expression $[o/x]e_i$, if $h(o) = (C, \perp)$ and $C \Downarrow \{C_1, C_2\} = C_i$.

In rules **SENDCASE-R** and **RECEIVECASE-R** it is understood that the rule cannot fire if $C \Downarrow \{C_1, C_2\} = \perp$. However we will see that $C \Downarrow \{C_1, C_2\}$ is always defined in well-typed expressions.

Rules **SENDWHILE-R** and **RECEIVEWHILE-R** simply realize the repetition of the case communication expressions. Note that $\text{sendW}(\mathcal{E})\{e_1 \Rightarrow C_1 \parallel e_2 \Rightarrow C_2\}$ is not reduced in an evaluation context, since we do not want to reduce the expression which controls the loop before the application of rule **SENDWHILE-R**, in which the sendW expression is unfolded.

Only communication and delegation expressions containing explicit channels can be reduced. So, for example, $\text{sendC}(o)\{e\}$ and $o \bullet s\}$ are stuck; however, as we saw in Subsection 6.1, the latter cannot be typed and the former is not an initial expression (type soundness is only guaranteed for initial expressions).

6 Typing

Session types, ranged over by \mathbf{t} , describe the communications that take place during a session. The syntax of session types is in Figure 9, where we use \dagger as a symbol that stands for either $!$ or $?$. By ε we denote the *empty* communication, and the concatenation $\mathbf{t}_1.\mathbf{t}_2$ expresses the communications in \mathbf{t}_1 followed by those in \mathbf{t}_2 . The session type ε is the neutral element of concatenation, so that $\varepsilon.\mathbf{t} = \mathbf{t} = \mathbf{t}.\varepsilon$ for all \mathbf{t} .

The types $!\{C_1 \Rightarrow \mathbf{t}_1 \parallel C_2 \Rightarrow \mathbf{t}_2\}$ and $?\{C_1 \Rightarrow \mathbf{t}_1 \parallel C_2 \Rightarrow \mathbf{t}_2\}$ express the session types for the sending and the receiving of an object, respectively: depending on the class of this object, the communication will proceed with one of the \mathbf{t}_i . In $\mu\alpha.\dagger\{C_1 \Rightarrow \mathbf{t}_1 \parallel C_2 \Rightarrow \mathbf{t}_2\}$ the *type variable* α can occur inside \mathbf{t}_i with the usual meaning of representing a recursive session type. We consider recursive session types modulo fold/unfold: *i.e.*,

² In this particular case, there is no other motivation for selecting the smallest index i for which C_i is the smallest class in $\{C_1, C_2\}$ to which the object o belongs, introducing non-deterministic choices. From this point of view, alternative solutions are possible, as long as they are just as sound: for instance, the selection of the greatest index or linguistic restrictions on the expressions e_i , *e.g.*, the condition $e_1 = e_2$ whenever $C_1 = C_2$.

$[\mu \alpha. \tau / \alpha] \tau$. So we equate $\mu \alpha. \dagger\{C_1 \Rightarrow \tau_1 \parallel C_2 \Rightarrow \tau_2\}$ to $\dagger\{C_1 \Rightarrow \tau_1 \parallel C_2 \Rightarrow \tau_2\}$ if $\mu \alpha. \tau$ does not occur in $\dagger\{C_1 \Rightarrow \tau_1 \parallel C_2 \Rightarrow \tau_2\}$.

The type \odot is used only as session type for the command `cont`: it plays the role of a placeholder which will be replaced by a type variable when the while expression is completed (see rules SENDW-T and RECEIVEW-T in Figure 11).

We say that a session type is *closed* if it does not contain occurrences of free type variables and of \odot . Therefore, each closed session type has one of the following shapes:

- ε ;
- $\mu \alpha. \dagger\{C_1 \Rightarrow \tau_1 \parallel C_2 \Rightarrow \tau_2\}$ or $\dagger\{C_1 \Rightarrow \tau_1 \parallel C_2 \Rightarrow \tau_2\}$;

or a concatenation of the session types above. For simplicity we will use in our examples unfolded recursive types whenever possible.

6.1 Typing of Channel Free Expressions

In this subsection we define typing for user expressions, in which communication channels are implicit. For technical reasons it is useful to consider also expressions containing occurrences of object identifiers, which are not directly expressible in user syntax. We call these expressions *channel free expressions*. The term environments then contain also type assignments to object identifiers. This permits a simpler formulation of the runtime typing rules, as we will see in next subsection.

The typing judgement has the shape

$$\Gamma \vdash e : T \circ \tau$$

where Γ is a term environment, which maps `this`, `cont`, variables and object identifiers to session types, and τ represents the session type of the (implicit) active channel.

In order to allow (possible) multiple occurrences of a variable or `cont` within expressions, we define the following “update” operation on Γ (over `this`, `cont`, term variables, and object identifiers):

$$\Gamma(z : T)(z') = \begin{cases} T & \text{if } z' = z \\ \Gamma(z') & \text{otherwise.} \end{cases}$$

Thus, the operation $\Gamma(z : T)$ has the effect of adding $z : T$ to Γ , but after removing the declaration of z from Γ (if there is one). This will avoid checking well-formedness of term environments and does not require an explicit weakening rule to add an assignment to the `cont` variable (when typing nested while communication expressions).

To assure a safe communication between two threads we must require the session types to be *dual*, i.e., that each send will correspond to a receive and vice versa. Duality is then the symmetric relation generated by the rules of Figure 10, in which we consider folded recursive types, otherwise the definition would not be well

$$\frac{C_1 \vee C_2 <: C_1 \vee C_2 \quad C_i \Downarrow \{C_1, C_2\} = C_j \Rightarrow \tau_i \bowtie \tau'_j \quad C_l \Downarrow \{C_1, C_2\} = C_k \Rightarrow \tau_k \bowtie \tau'_k}{\mu\alpha.!\{C_1 \Rightarrow \tau_1 \parallel C_2 \Rightarrow \tau_2\} \bowtie \mu\alpha.?\{C'_1 \Rightarrow \tau'_1 \parallel C'_2 \Rightarrow \tau'_2\}}$$

Fig. 10. Duality Relation

The exchanged values must also be of one of the classes expected by the receiver. The possible choices on the basis of the class of the exchanged value must conform to the session types which are dual of each other. For this reason we have to perform a check on the type of the exchanged values in both directions:

- for any sent value of type C_i such that $C_i \Downarrow \{C'_1, C'_2\} = C'_j$ for some $1 \leq j \leq 2$ require $\tau_i \bowtie \tau'_j$;
- for any received value of type C'_l such that $C'_l \Downarrow \{C_1, C_2\} = C_k$ for some $1 \leq k \leq 2$ we require $\tau_k \bowtie \tau'_l$.

For instance, let us consider the session types $!\{\text{Shape} \Rightarrow \tau_1 \parallel \text{String} \Rightarrow \tau_2\} \bowtie ?\{\text{Triangle} \Rightarrow \tau_3 \parallel \text{Object} \Rightarrow \tau_4\}$ where $\text{Triangle} <: \text{Shape}$. At run time, Triangle can be sent as a Shape, thus the types τ_1 and τ_3 have to be dual. This is possible that, thanks to the absence of generics we can be more flexible w.r.t. [6]: the types in the choices (actually their union) of the send can be subtypes of the ones in the dual receive).

Typing rules for channel free expressions are in Figure 11. For the sake of simplicity, in rule NEWC-T we require that the initialisation of an object does not involve communications. Notice that in rule SEQ-T we use session type concatenation to express that first the communications in e_1 and then those in e_2 are performed.

The rule for session request SESSREQ-T relies on the duality relation (Fig. 10) to assure that all the bodies of the session s in the classes which build the union type T and the co-body e' of the request will communicate properly. In typing session delegation (rule SESSDEL-T) we take into account that the whole expression e is replaced by the session body defined in the class of the expression to which the delegation is delegated (cf. the reduction rule SESSDEL-R, Figure 8). Notice that the condition $\text{stype}(s, T) = \{\tau'\}$ does not imply T be one class, but only that all definitions of T are classes which build T have the same session types. If a session has session type τ' it is meaningless to use it in a delegation, while it is sensible to use it in a request. For this reason we require $\tau' \neq \varepsilon$ in rule SESSDEL-T.

Rules SENDC-T and RECEIVEC-T require all possible alternative expressions to have the same type T , but they can implement different communication sequences. Rule SENDC-T prescribes that the class type of e is the union type of the classes in the choice. Without union types the typing rule for the same construct in [6] is much more demanding and less clear. The typing rules for the while communication expressions are similar, but they also discharge the assumption on cont and res . The occurrences of \odot in session types by a fresh variable α which will be bound in rule SENDW-T typing e with session type ε prevents e from containing occurrences of \odot .

$$\begin{array}{c}
\text{NEWC-T} \\
\frac{\text{fields}(C) = \overline{Tf} \quad \Gamma \vdash e_i : T_i \wp \varepsilon}{\Gamma \vdash \text{newC}(\overline{e}) : C \wp \varepsilon} \\
\\
\text{SEQ-T} \\
\frac{\Gamma \vdash e : T \wp t \quad \Gamma \vdash e' : T' \wp t'}{\Gamma \vdash e; e' : T' \wp t.t'} \\
\\
\text{FLD-T} \\
\frac{\Gamma \vdash e : T \wp t}{\Gamma \vdash e.f : \text{ftype}_r(f, T) \wp t} \\
\\
\text{FLDASS-T} \\
\frac{\Gamma \vdash e : T \wp t \quad \Gamma \vdash e' : \text{ftype}_w(f, T) \wp t}{\Gamma \vdash e.f := e' : \text{ftype}_w(f, T) \wp t.t'} \\
\\
\text{SESSREQ-T} \\
\frac{\Gamma \vdash e : T \wp t \quad \Gamma \vdash e' : T' \wp t' \quad t' \bowtie t'' \forall t'' \in \text{stype}(s, T)}{\Gamma \vdash e.s\{e'\} : T' \wp t} \\
\\
\text{SESSDEL-T} \\
\frac{\Gamma \vdash e : T \wp t \quad \text{stype}(s, T) = \{t'\} \quad t' \neq \varepsilon \quad \text{rtype}(s, T) = T'}{\Gamma \vdash e \bullet s\{t'\} : T' \wp t.t'} \\
\\
\text{SEND C-T} \\
\frac{\Gamma \vdash e : C_1 \vee C_2 \wp t \quad \Gamma \vdash e_i : T \wp t_i}{\Gamma \vdash \text{sendC}(e)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \wp t.!\{C_1 \Rightarrow t_1 \parallel C_2 \Rightarrow t_2\}} \\
\\
\text{RECEIVE C-T} \\
\frac{\Gamma(x : C_i) \vdash e_i : T \wp t_i}{\Gamma \vdash \text{recC}(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \wp ?\{C_1 \Rightarrow t_1 \parallel C_2 \Rightarrow t_2\}} \\
\\
\text{SEND W-T} \\
\frac{\Gamma \vdash e : C_1 \vee C_2 \wp \varepsilon \quad \Gamma(\text{cont} : T) \vdash e_i : T \wp t_i \quad \alpha \text{ fresh in } t_1, t_2}{\Gamma \vdash \text{sendW}(e)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \wp \mu\alpha.!\{C_1 \Rightarrow [\alpha/\odot]t_1 \parallel C_2 \Rightarrow [\alpha/\odot]t_2\}} \\
\\
\text{RECEIVE W-T} \\
\frac{\Gamma(\text{cont} : T)(x : C_i) \vdash e_i : T \wp t_i \quad \alpha \text{ fresh in } t_1, t_2}{\Gamma \vdash \text{recW}(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \wp \mu\alpha.?\{C_1 \Rightarrow [\alpha/\odot]t_1 \parallel C_2 \Rightarrow [\alpha/\odot]t_2\}}
\end{array}$$

Fig. 11. Typing Rules for Channel Free Expressions

of communications and cont^3 . Notice that requiring that both branch expressions of the choice operation have the same union type T does not imply that we require t_1 and t_2 of the same class: in fact, T can be a proper union type. For instance, we may have $\Gamma \vdash e_1 : T_1 \wp t_1$ and $\Gamma \vdash e_2 : T_2 \wp t_2$; by subsumption (rule SUB-T) we also have $\Gamma \vdash e_1 : T_1 \vee T_2 \wp t_1$ and $\Gamma \vdash e_2 : T_1 \vee T_2 \wp t_2$. Then, $T = T_1 \vee T_2$.

Figure 12 defines well-formed class tables. Rule SESS-WF type checks the session types of the bodies with respect to the current class C taking as term environment the one defined by the bodies between this and C . Notice that \odot has no dual type, so sessions whose bodies are typed with types containing \odot would be useless. This justifies the condition that such sessions must be closed in rule SESS-WF.

A last remark is that, since no typing rule generates free session type variables, all session types in typing judgements are closed unless they contain occurrences of

³ Note that this typing allows e to contain session requests, since the execution of the bodies will use different channels to communicate.

The rules, presented in this section, define how the type system checks the declarative part of the program and the main part are well-typed, also with session types used in declarations of sessions. However, when considering well-typed executable programs, we require that they are closed with respect to term variables, that all communication expressions occur inside session co-bodies, that is, that they are typed in the empty type environment with an empty session type. Namely, an expression e is such that $\emptyset \vdash e : T \sharp \varepsilon$ for some T . It is easy to verify that the set of initial expressions is the set of closed and well-typed user expression. For example, we consider the stuck expressions $\text{sendC}(o)\{e\}$ and $o \bullet s\{\}$:

- $\text{sendC}(o)\{e\}$ is well-typed but its session type is not empty,
- $o \bullet s\{\}$ is not well-typed.

We conclude this subsection by comparing $\mathcal{F}\text{SAM}^\vee$ with $\text{FJ}\vee$, an extension of FJ with union types, proposed by Igarashi and Nagira in [29]. They define union types in the present paper: the essential difference is that they have traditional method signatures for sessions.

The method signatures are of the shape $\bar{T} \rightarrow T$, where both the parameter type \bar{T} and the return type T are union types. The method type lookup function applied to a name m and to a union type T gives a set of method signatures, *i.e.* all the signatures which m has in the classes which build T . This is similar to our *stype* function which returns a set of session types.

The rule of method call checks that the types of the parameters agree with the signatures found by the method type lookup function for the union type of the callee. Also our rule SESSREQ-T requires the session type of the co-body be dual to the session types returned by the *stype* function.

It is easy to check that the encoding of methods by sessions sketched at the end of Section 3 extends without changes to methods with union types.

6.2 Typing of Runtime Expressions

During evaluation of well-typed programs, channel names are made explicit in receive expressions, as well as in session delegation. Thus, in order to show that well-typedness is preserved under evaluation, we need to define new typing rules for runtime expressions. Furthermore, in typing runtime expressions, we must take into account session types of more than one channel: runtime expressions contain explicit channel names (used for communication) thus session types must be associated with these names in an appropriate way. Then judgements have the form

$$\Gamma \vdash_{\Sigma} e : T \sharp \Sigma$$

where Σ denotes a *session environment* which maps channels to session types.

NEWC-RT $\frac{\text{fields}(C) = \overline{Tf}}{\Gamma \vdash_{\text{R}} \text{newC}(\overline{e}) : C \circ \emptyset}$	CONT-RT $\frac{}{\Gamma(\text{cont} : T) \vdash_{\text{R}} \text{cont} : T \circ \{k : \odot\}}$
SEQ-RT $\frac{\Gamma \vdash_{\text{R}} e : T \circ \Sigma \quad \Gamma \vdash_{\text{R}} e' : T' \circ \Sigma'}{\Gamma \vdash_{\text{R}} e; e' : T' \circ \Sigma. \Sigma'}$	FLDASS-RT $\frac{\Gamma \vdash_{\text{R}} e : T \circ \Sigma \quad \Gamma \vdash_{\text{R}} e' : \text{ftype}_w(f, T)}{\Gamma \vdash_{\text{R}} e.f := e' : \text{ftype}_w(f, T) \circ \Sigma. \Sigma'}$
SESSREQ-RT $\frac{\Gamma \vdash_{\text{R}} e : T \circ \Sigma \quad \Gamma \vdash e' : T' \circ t' \quad t' \bowtie t'' \forall t'' \in \text{stype}(s, T)}{\Gamma \vdash_{\text{R}} e.s\{e'\} : T' \circ \Sigma}$	
SESSDEL-RT $\frac{\Gamma \vdash_{\text{R}} e : T \circ \Sigma \quad \text{stype}(s, T) = \{t\} \quad t \neq \varepsilon \quad \text{rtype}(s, T) = T'}{\Gamma \vdash_{\text{R}} e \bullet s\{k\} : T' \circ \Sigma.\{k : t\}}$	
SENDC-RT $\frac{\Gamma \vdash_{\text{R}} e : C_1 \vee C_2 \circ \Sigma \quad \Gamma \vdash_{\text{R}} e_i : T \circ \{k : t_i\}}{\Gamma \vdash_{\text{R}} k.\text{sendC}(e)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \circ \Sigma.\{k : !\{C_1 \Rightarrow t_1 \parallel C_2 \Rightarrow t_2\}\}}$	
RECEIVEC-RT $\frac{\Gamma(x : C_i) \vdash_{\text{R}} e_i : T \circ \{k : t_i\}}{\Gamma \vdash_{\text{R}} k.\text{recC}(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \circ \{k : ?\{C_1 \Rightarrow t_1 \parallel C_2 \Rightarrow t_2\}\}}$	
SENDW-RT $\frac{\Gamma \vdash_{\text{R}} e : C_1 \vee C_2 \circ \emptyset \quad \Gamma(\text{cont} : T) \vdash_{\text{R}} e_i : T \circ \{k : t_i\} \quad \alpha \text{ fresh in } t_1, t_2}{\Gamma \vdash_{\text{R}} k.\text{sendW}(e)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \circ \{k : \mu\alpha.!\{C_1 \Rightarrow [\alpha/\odot]t_1 \parallel C_2 \Rightarrow [\alpha/\odot]t_2\}\}}$	
RECEIVEW $\frac{\Gamma(\text{cont} : T)(x : C_i) \vdash_{\text{R}} e_i : T \circ \{k : t_i\} \quad \alpha \text{ fresh in } t_1, t_2}{\Gamma \vdash_{\text{R}} k.\text{recW}(x)\{C_1 \Rightarrow e_1 \parallel C_2 \Rightarrow e_2\} : T \circ \{k : \mu\alpha.?\{C_1 \Rightarrow [\alpha/\odot]t_1 \parallel C_2 \Rightarrow [\alpha/\odot]t_2\}\}}$	

Fig. 13. Typing Rules for Runtime Expressions

A session environment maps only a finite set of channels to session types from ε , and all the remaining to ε . We can then represent one session environment as an infinite number of finite sets which give all the meaningful associations and the others. For example $\{k : t\}$ and $\{k : t, k' : \varepsilon\}$ represent the same environment. This choice avoids an explicit weakening rule for session environments. Figure 13 shows the typing rules for runtime expressions, which differ from those for channel expressions for having session environments instead of a unique session type. For the concatenation of session types we extend the *concatenation* of session types to *session environments* as follows.

$$\Sigma. \Sigma'(k) = \Sigma(k). \Sigma'(k)$$

Notice that in rule SESSREQ-RT we are making use of the judgement $\Gamma \vdash e' : T' \circ t'$ where the expression e' does not contain channels, but it can contain objects and variables. This justifies our choice of considering channel free expressions instead of expressions in the typing rules of previous subsection. Notice also that the session environments of the branches in the communication expressions only contain the channel as subject, since these expressions will never be reduced before the

6.3 Type Soundness

Our type system enjoys subject reduction and assures progress (\longrightarrow^* is the reflexive and transitive closure or \longrightarrow):

If $\emptyset \vdash e_0 : T_0 \S \varepsilon$ and $e_0, [] \longrightarrow^* P, h$, where $P \equiv e_1 \parallel \dots \parallel e_n$, then:

- for each e_i we get $\Gamma \vdash_r e_i : T_i \S \Sigma_i$ for some Γ, T_i, Σ_i ($1 \leq i \leq n$), and $(1 \leq j \leq n)$ such that $T_j = T_0$, and;
- either $P, h \longrightarrow P', h'$ for some P', h' , or for all i ($1 \leq i \leq n$) e_i is an object.

The runtime errors which our type system prevents are:

1. the selection of a field and the request of a session which do not belong to the object of the current object;
2. the creation of a pair of dual channels whose communication sequences do not perfectly match.

Proofs, more examples and discussions can be found in the extended version of this paper, available at <http://www.di.unito.it/~dezani/papers/bcdgfull.pdf>.

7 Conclusion

In the present paper we showed, through the language SAM^\vee , how the addition of session types to an object oriented language with session types enhances flexibility.

The amalgamation of communication centred and object oriented programming paradigms it has been developed in [18, 6] and in the present paper, can be extended in various directions. In particular we plan to integrate this approach with multi-party session types [9] and with safe failure recovery [4].

Moreover, we want to study the extension of union and intersection to session types, following the intuition given by union/intersection of contracts in [12].

Lastly it would be interesting to integrate session primitives with name calculi as introduced in [5] in order to allow specification of Quality of Service requirements.

Acknowledgements. We thank the referees for their helpful comments. The final version of the paper improved due to their suggestions.

References

1. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Intersection and Union Types and Semantics. *Information and Computation* 119, 202–230 (1995)
2. Bonelli, E., Compagnoni, A.: Multipoint Session Types for a Distributed Calculus. In: G., Fournet, C. (eds.) *TGC 2007. LNCS*, vol. 4912, pp. 240–256. Springer, Heidelberg, appear, 2008)
3. Bonelli, E., Compagnoni, A., Gunter, E.: Correspondence Assertions for Process Calculi and Type Theory. *Journal of Functional Programming* 15(2), 1–40 (2005)

- pp. 38–57. Springer, Heidelberg (2006)
5. Buscemi, M., Montanari, U.: CC-Pi: A Constraint-Based Language for Service Level Agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
6. Capecchi, S., Coppo, M., Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, M.: Amalgamating Sessions and Methods in Object Oriented Languages with Generics. In: ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
7. Carbone, M., Honda, K., Yoshida, N.: A Calculus of Global Interaction Based Types. In: Fernández, M., Kirchner, C. (eds.) SecReT 2006. ENTCS, vol. 171(3), pp. 151–175. Elsevier, Amsterdam (2007)
8. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
9. Carbone, M., Honda, K., Yoshida, N.: Multiparty Asynchronous Session Types. In: G.C., Wadler, P. (eds.) POPL 2008, pp. 273–284. ACM Press, New York (2008)
10. Castagna, G., De Nicola, R., Varacca, D.: Semantic Subtyping for the π -calculus. In: Gaden, P. (ed.) LICS 2005, pp. 92–101. IEEE Computer Society Press, Los Alamitos (2005)
11. Castagna, G., Frisch, A.: A Gentle Introduction to Semantic Subtyping. In: Barthe, G., Felty, A.P. (eds.) PPDP 2005, pp. 198–199. ACM Press, New York (2005)
12. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. In: G.C., Wadler, P. (eds.) POPL 2008, pp. 261–272. ACM Press, New York (2008)
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and their Application to Object-Oriented Languages. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOF 2007. LNCS, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
14. Dezani-Ciancaglini, M., de’ Liguoro, U., Yoshida, N.: On Progress for Structured Communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 1–17. Springer, Heidelberg (2008)
15. Dezani-Ciancaglini, M., Drossopoulou, S., Giachino, E., Yoshida, N.: Bounded Session Types for Object-Oriented Languages. In: de Boer, F., Bonsangue, M. (eds.) FM 2007. LNCS, vol. 4709, pp. 207–245. Springer, Heidelberg (2007)
16. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4066, pp. 325–352. Springer, Heidelberg (2006)
17. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A Distributed Object-Oriented Language with Session Types. In: De Nicola, R., Sangiorgi, D. (eds.) ESOP 2005. LNCS, vol. 3705, pp. 299–318. Springer, Heidelberg (2005)
18. Drossopoulou, S., Dezani-Ciancaglini, M., Coppo, M.: Amalgamating the Session Types and the Object Oriented Programming Paradigms. In: MPOOL 2007 (2007), <http://homepages.fhn-regensburg.de/mpool/mpool07/programme.html>
19. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R.: Language Support for Fast and Reliable Message-based Communication in Singa. In: Zwaenepoel, W. (ed.) EuroSys 2006. ACM SIGOPS, pp. 177–190. ACM Press, New York (2006)
20. Fournet, C., Laneve, C., Maranget, L., Rémy, D.: Inheritance in the Join Calculus. In: S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 397–408. Springer, Heidelberg (2000)
21. Gapeyev, V., Pierce, B.C.: Regular Object Types. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 151–175. Springer, Heidelberg (2003)

24. Gay, S., Hole, M.: Subtyping for Session Types in the π -Calculus. *Acta Informatica* 36(3), 191–225 (2005)
25. Gay, S., Vasconcelos, V.T., Ravara, A.: Session Types for Inter-Process Communication. In: *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 2003–133, Department of Computing, University of Glasgow (2003)
26. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) *CONCUR 1993. LNCS*, vol. 78, pp. 509–523. Springer, Heidelberg (1993)
27. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-based Programming. In: Hankin, C. (ed.) *ESOP 1998. LNCS*, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)
28. Honda, K., Yoshida, N., Carbone, M.: Web Services, Mobile Processes and Type Theory. *Journal of Functional Programming* 17(1), 160–188 (2007)
29. Igarashi, A., Nagira, H.: Union Types for Object Oriented Programming. *Journal of Functional Programming* 6(2), 31–52 (2007)
30. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a Minimal Core Calculus for the Java Virtual Machine. *ACM TOPLAS* 23(3), 396–450 (2001)
31. Sparkes, S.: Conversation with Steve Ross-Talbot. *ACM Queue* 4(2), 14–23 (2000)
32. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing Theory. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) *PARLE 1994. LNCS*, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
33. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the Behavior of Objects and Processes using Session Types. In: Brogi, A., Jacquet, J.-M. (eds.) *FOCLASA 2000. LNCS*, vol. 68(3), pp. 439–456. Elsevier, Amsterdam (2002)
34. Vasconcelos, V.T., Gay, S., Ravara, A.: Typechecking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science* 368(1-2), 64–87 (2006)
35. Web Services Choreography Working Group. Web Services Choreography Description Language (2002), <http://www.w3.org/2002/ws/chor/>